# Parallel Turing Machines With One-Head Control Units And Cellular Automata*

Thomas Worsch†
http://liinwww.ira.uka.de/~worsch/

**Abstract**

*Parallel Turing machines* (PTM) can be viewed as a generalization of cellular automata (CA) where an additional measure called processor complexity can be defined which indicates the "amount of parallelism" used. In this paper PTM are investigated with respect to their power as recognizers of formal languages. A combinatorial approach as well as diagonalization are used to obtain hierarchies of complexity classes for PTM and CA. In some cases it is possible to keep the space complexity of PTM fixed. Thus for the first time it is possible to find hierarchies of complexity classes (though not CA classes) which are completely contained in the class of languages recognizable by CA with space complexity $n$ and in polynomial time. A possible collapse of the time hierarchy for these CA would therefore also imply some unexpected properties of PTM.

*Key words:* cellular automata, parallel Turing machines, computational complexity, theory of parallel computation

## 1 Introduction

Since their introduction by von Neumann cellular automata have become a well known model of computation. In this paper we are interested in their power as recognizers of formal languages. Some results from complexity theory will be presented which are related to earlier work by Ibarra, Kim and Moran [8], but will be stated for the more general model of PTM. They can also be taken as an indication of what might be the solution for a still open problem for cellular automata (see [7]):

Consider one-dimensional CA working in linear space, i.e., for inputs of length $n$ the CA use $n$ cells. The largest time complexities of these automata are of the form $c^n$. But until now it is not known, whether there is a formal language which can be recognized by CA in linear space, and which *requires* an exponential or at least a non linear time complexity. This is in contrast to Turing machines, where a dense hierarchy of time complexity classes between $n$ and $n^2$ — even for a fixed space complexity of $n$ — is known [6]. Hennie's proof is a combinatorial one. It is usually difficult to use diagonalization constructions if the space complexity is fixed.

In order to find out more about CA, the generalized model of *parallel Turing machines* is investigated. Although the original problem cannot be solved, at least some partial answers as well as related results for PTM are obtained.

Throughout the paper we will consider one-dimensional CA and PTM. Such a PTM consists of one one-dimensional tape, on which a (possibly varying) number of Turing control units with one head each, i.e., finite automata, is working cooperatively. Besides the usual complexity measures space and time, one can consider the maximum number of finite automata which simultaneously exist during a computation. This "processor complexity" allows one to distinguish in a formal way CA, for which one has the intuitive impression that

*This is technical report 3/97 of the Department of Informatics, University of Karlsruhe

†Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany.

one of them "makes more use of parallelism" than the other, although their time and space complexities are identical. For example all known fast algorithms for the recognition of palindromes (and "similar" languages) require a lot of "activities" whereas $\{0^n1^n2^n \mid n \in \mathbb{N}_+\}$ only requires very few. Furthermore it seems that in the case of palindromes a lot of activities are *necessary* for fast recognition. PTM allow a precise statement of such facts and their proofs.

The version of PTM we are interested in in this paper has first been introduced by Hemmerling [3, 4] under the name of *systems of Turing automata*. He was interested in the general $d$-dimensional case and shows the equivalence of the realizability of synchronization, concentration and certain pattern transformations for classes of $d$-dimensional patterns.

Later Wiedermann generalized systems of Turing automata to so called *parallel Turing machines* [17, 18] allowing several read-write heads per control unit and several tapes. In this case care has to be taken in order to devise a sensible definition of space complexity [20]. Wiedermann sketches a simulation of PTM by CA but does not further investigate the relations between the two.

Another formalization of the concept of the different amounts of "activities" in CA is the so called state change complexity, introduced by Vollmar [14] and further investigated by Suel [13]. It is possible to prove the separation of specific complexity classes by combinatorial arguments using state change complexity, but no large hierarchies are known.

The rest of this paper is organized as follows: In Section 2, we give the basic definitions, mainly for PTM and their complexity measures. In Section 3 basic tools for the construction of PTM are introduced, which also deserve some interest on their own. Section 4 is concerned with tradeoffs between time and processor complexity in general and in particular for the recognition of a certain language. These results are used in the following two sections. In Section 5 connections between PTM, TM and CA are investigated. In Section 6 the existence of arbitrarily large finite hierarchies of (time and processor) complexity classes is proved for fixed space complexity. Sections 7 and 8 are devoted to two diagonalization constructions. In the first case the processor complexity is fixed and as a special case very fine time hierarchies for CA are obtained. In the second case we show how to keep the space complexity fixed at the expense of an increasing processor complexity.

# 2 Basic Notions

## 2.1 Parallel Turing Machines

The following definition of PTM is almost the same as the one given by Hemmerling [3], but differs from Wiedermann's [17]. The basic concepts are the same in both cases, but it is our impression that the definition given below is a little bit more convenient for the description of concrete algorithms.

**2.1 Definition.** *A* parallel Turing machine *consists of a usual one-dimensional Turing tape, on which a number of finite automata are working. It is characterized by an 8-tuple* $P = (Q, q_0, F_+, F_-, B, A, \square, \delta)$. *$Q$ is the* set of states *and contains an initial state $q_0$. The disjoint subsets $F_+$ and $F_-$ of $Q$ contain the* accepting *resp.* rejecting final states. *It is required that $q_0 \notin F_+ \cup F_-$. $B$ is the* tape alphabet *containing at least the blank symbol $\square$ and the symbols of the* input alphabet *$A$.*

*A* configuration *of a* PTM $P = (Q, \ldots)$ *is a pair $c = (p, b)$ of mappings*[1] $p : \mathbb{Z} \to 2^Q$ *and $b : \mathbb{Z} \to B$, where $p(i)$ is the set of states of the finite automata currently visiting square $i$ and $b(i)$ is the symbol written on it.* [2]

*Each step of a* PTM, *i.e. the transition from a configuration $c$ to its successor configuration $c' = (p', b')$ is determined by the* transition function $\delta : 2^Q \times B \to 2^{Q \times D} \times B$ *where $D$ is the set $\{-1, 0, 1\}$ of possible movements of a finite automaton. In order to compute $c'$, $\delta$ is*

[1] $2^Q$ denotes the power set of $Q$.

[2] This means that it is not possible to have two automata on the same square and in the same state simultaneously.

*simultaneously applied at all tape positions $i \in \mathbb{Z}$. The arguments used are the set of states of the finite automata currently visiting square $i$ and its tape symbol. Let $(M'_i, b'_i) := \delta(p(i), b(i))$. Then the new symbol on square $i$ in configuration $c'$ is $b'(i) := b'_i$. The set of finite automata on square $i$ is replaced by a new set of finite automata (defined by $M'_i \subseteq Q \times D$) each of which has to change the tape square according to the indicated direction of movement, i.e., $p'(i) := \{q \mid (q,1) \in M'_{i-1} \vee (q,0) \in M'_i \vee (q,-1) \in M'_{i+1}\}$.*

Observe that the number of finite automata on the tape may change during a computation. Automata may vanish (if for example[3] $\delta(\{s\}, b)[1] = \emptyset$) and new automata may be generated (if for example $\delta(\{s\}, b)[1] = \{(q,1), (q',0)\}$). But in order to make the model useful (and to come up to some intuitive expectations) it is required, that automata cannot arise from "nothing" and that the symbol on a tape square can only change, if it is visited by at least one finite automaton. In other words: $\forall\, b \in B : \delta(\emptyset, b) = (\emptyset, b)$.

A tape square $i$ is called *empty* in a configuration $c = (p, b)$ if $p(i) = \emptyset$, and it is called *blank* if $b(i) = \Box$. A tape square is *used* if it is neither empty nor blank.

Sometimes we will speak of a *cell* of a PTM. By that we mean a pair $(R, b)$ consisting of the set of states $R$ of the finite automata currently visiting a tape square and the symbol $b$ written on it.

For the recognition of formal languages we define the *initial configuration* $c_w$ for an input word $w \in A^+$ as the one in which $w$ is written on the otherwise blank tape on squares $1, 2, \ldots, |w|$, and in which there exists exactly one finite automaton in state $q_0$ on square 1.

A configuration $(p, b)$ of a PTM is called *accepting* iff $p(1) \subseteq F_+$, and it is called *rejecting* iff $p(1) \subseteq F_-$. Accepting and rejecting configurations are also referred to as final ones. As usual the *language recognized by a* PTM $P$ is the set of input words, for which the first final configuration reached is an accepting one.

Several other possibilities for the definition of initial and final configurations also have been investigated and can be shown to be essentially equivalent with respect to the complexity measures defined below (see [19]).

In the rest of this paper we restrict ourselves to PTM, which reach a final configuration for every input word. Hence the following three functions are all total ones.

**2.2 Definition.** *For a* PTM $P$ *and an input word* $w$ $\mathrm{time}_P(w)$ *denotes the number of steps $P$ makes starting from $c_w$ before reaching a final configuration for the first time. The* time complexity *of $P$ is* $\mathrm{Time}_P : \mathbb{N}_+ \to \mathbb{N}_+ : n \mapsto \max\{\mathrm{time}_P(w) \mid w \in A^n\}$. *Similarly let* $\mathrm{space}_P(w)$ *denote the number of squares used by $P$ during the computation for $w$. The* space complexity *of $P$ is* $\mathrm{Space}_P : \mathbb{N}_+ \to \mathbb{N}_+ : n \mapsto \max\{\mathrm{space}_P(w) \mid w \in A^n\}$. *And we write* $\mathrm{proc}_P(w)$ *for the maximum number of finite automata which exist in a configuration occuring in the computation for input $w$ and define the* processor complexity *of $P$ as* $\mathrm{Proc}_P : \mathbb{N}_+ \to \mathbb{N}_+ : n \mapsto \max\{\mathrm{proc}_P(w) \mid w \in A^n\}$.

Obviously, the relation $1 \leq \mathrm{Proc}_P(n) \leq |Q| \cdot \mathrm{Space}_P(n)$ holds for all $n \in \mathbb{N}_+$. It follows immediately from Theorem 5.2 below, that the processor complexity is not a Blum measure. Nevertheless it can be used for finding additional structure within already restricted complexity classes.

For total functions $s$, $t$ and $h$ from $\mathbb{N}_+$ into $\mathbb{N}_+$, we define the complexity class PTM–STP$(s, t, h)$ to be the family of all languages $L$ for which there is a PTM recognizing $L$ and satisfying, for all $n \in \mathbb{N}_+$, $\mathrm{Space}_P(n) \leq s(n)$, $\mathrm{Time}_P(n) \leq t(n)$, and $\mathrm{Proc}_P(n) \leq h(n)$. We also use PTM–ST$(s, t)$, and so on. Furthermore we write PTM–T(O$(t)$) instead of $\bigcup_{t' \in \mathrm{O}(t)}$ PTM–T$(t')$ and so on.

## 2.2 Turing Machines and Cellular Automata

We assume that readers are familiar with Turing machines. The version used in this paper has one or several one-dimensional work tapes and a control unit with one or several read-write heads on each of the tapes. When we speak of (1)–TM we think of the special case of one

[3] The $i$-th component of vector $\mathbf{v}$ is denoted by $\mathbf{v}[i]$, i.e. $\mathbf{v} = (\mathbf{v}[1], \ldots, \mathbf{v}[|v|])$.

tape and a control unit with only one head on this tape. At the beginning of a computation the input word is always stored on the first of the tapes; all other tapes (if any) are initially blank.

We will consider one-dimensional cellular automata using (w.l.o.g.) von Neumann neighborhood $N = \{-1, 0, 1\}$. If $Q$ denotes the set of states, the local rule is of the form $\delta : Q^N \to Q$. For a global configuration $c : \mathbb{Z} \to Q$ and $x \in \mathbb{Z}$ let $c_x : N \to Q : n \mapsto c(x+n)$ denote the local configuration observed in $c$ at cell $x$. A global transition step leads from configuration $c$ to $\Delta(c)$ where $\Delta$ is defined by: $\forall x \in \mathbb{Z} : \Delta(c)(x) = \delta(c_x)$. We assume that the input alphabet $A$ is a subset of $Q$ and that in the initial configuration $c_w$ for an input $w \in A^+$ the cell $i$, $1 \leq i \leq |w|$, stores the input symbols $w[i]$ and all other cells are in a quiescent state (which they may leave if at least one neighbor is non-quiescent). An input is accepted (rejected) if at some time the cell 1 enters an accepting (rejecting) final state.

Complexity classes for cellular automata are denoted as CA–ST$(s,t)$. In the case of Turing machines with possibly several tapes and several heads on each tape we write TM–ST$(s,t)$. If only Turing machines with one tape and one head are considered, we speak of (1)-Turing machines and write e.g. (1)–TM–ST$(s,t)$.

# 3 Basic Tools

## 3.1 Synchronization

It will turn out in Section 5 that PTM are quite closely related to CA. It is therefore reasonable to consider the following generalization of the Firing Squad Synchronization Problem (FSSP) which is a famous problem for cellular automata (e.g. [9]):

**3.1 Problem. (FSSP for PTM)** *On some squares of a finite segment of the tape of a PTM non-moving finite automata are positioned. The leftmost and rightmost ones are in special states designating them as borders. The task is to achieve that at some point of time all finite automata simultaneously for the first time enter a special ("firing") state and that no additional automata are present.*

If one would be interested only in time optimal synchronization, one could of course fill the gaps between the finite automata to be synchronized with additional ones and then use the chain as a cellular automaton for an FSSP algorithm (see the special case of Theorem 5.1 for $h = s$). When the time point of firing is reached, the additional automata are simply deleted. But in that way the processor complexity may be increased by an arbitrarily large amount (e.g. in the case of only two finite automata to be synchronized, which are arbitrarily far away). This would limit the range of applications where a solution to the FSSP can be used (e.g. it would not be appropriate for the proof of Lemma 4.2 below). Fortunately it is possible to solve the problem in such a way that the number of processors used during the algorithm is only a constant times the number of finite automata to be synchronized.

**3.2 Lemma.** *There is a constant $c$ such that the FSSP for PTM can be solved in a time $2n-2$ using $cm$ processors where $n$ is the distance between left- and rightmost automaton and $m$ is the number of automata to be synchronized.*

**Proof.** Balzer [1] has suggested a $3n$ time FSSP algorithm for cellular automata. It uses two signals to divide the chain of cells to be synchronized into two halves. The signals meet in the middle and trigger the recursive application of the algorithm to both parts.

In a PTM the signals can be realized by two finite automata; the middle is marked by an additional automaton which is deleted at the firing if necessary. In addition to the original algorithm each of the signal automata checks, whether in "its half" there really is one of the finite automata to be synchronized. Only if this is the case, the synchronization is triggered on the corresponding half.

There is an FSSP algorithm for CA by Gerken [2] working in optimal time, i.e. $2n-2$ steps, to which the same technique can be applied. This is possible because Gerken's algorithm

also works by dividing the "area of synchronization" into parts on which the algorithm is applied recursively, but it uses much less "signals" than other time optimal FSSP algorithms. ∎

### 3.2 Constructibility

In several of the following sections there will be theorems which can only be proved if some complexity bounds involved "behave nicely" with respect to PTM. We therefore introduce some notion of PTM *constructibility* for functions.

**3.3 Definition.** *Let $s$, $t$, $h$ and $f$ be functions from $\mathbb{N}_+$ into $\mathbb{N}_+$, where $f(n) \geq 2$. We call $f$* fully PTM processor constructible *in space $s$, time $t$ and with $h$ processors iff there is a* PTM *$P = (Q, \ldots)$ having the following properties:*

- *$Q$ contains the 5 states $s_l, s_M, s_g, s_m$, and $s_r$.*
- *For each $w \in A^+$ starting from $c_w$ $P$ reaches a configuration $c'_w$ with the same tape inscription as $c_w$ and with exactly $f(|w|)$ finite automata distributed over $s(|w|)$ squares of the tape in such a way that for the $f(|w|)-1$ segments of empty tape squares between them holds:*
  - *The lengths of any two segments differ by at most one and all longer segments are to the left of all shorter segments.*
  - *The leftmost automaton is on square 1 in state $s_l$ and the rightmost automaton is on square $s(|w|)$ in state $s_r$.*
  - *If all segments are of the same length all automata between $s_l$ and $s_r$ are in state $s_M$.*
  - *If there are segments of two different lengths, the only automaton positioned between a longer and a shorter segment is in state $s_g$. The automata (if any) between $s_l$ and $s_g$ are in state $s_M$ and the automata (if any) between $s_g$ and $s_r$ are in state $s_m$.*
- *For all $n \in \mathbb{N}_+$ for all $w \in A^n$ in order to compute $c'_w$ from $c_w$ $P$ needs at most $s(n)$ tape squares, at most $t(n)$ steps, and at most $h(n)$ finite automata.*

*A function $s$ is called* fully PTM space constructible *in time $t$ and with $h$ processors iff the constant function 2 is fully* PTM *processor constructible in space $s$, time $t$ and with $h$ processors. In other words, the constructed space segment is marked by two automata at the left and right end of it.*

## 4 Tradeoffs between time and processor complexity

We start with results on possible slow-downs as consequences of a reduction of the number of processors. In the second subsection, we prove for a specific language that there is a tradeoff between the time needed and the number of processors used, which is close to the optimum. This will be exploited in a later section.

### 4.1 General Tradeoffs

**4.1 Lemma.** *Decreasing the processor complexity of a* PTM *by 1 can force an increase of the time complexity by a factor of $\log n$.*

**Proof.** The language $L = \{0^n 1^n \mid n \in \mathbb{N}_+\}$ can be recognized by a PTM with processor complexity 2 in linear time by simulating the standard idea for a CA recognizing $L$ (e.g. [15]): At the left end of the input two automata $A_1$ and $A_2$ are started moving to right with speeds 1/3 and 1. $A_2$ reverses its direction when it reaches the first blank square to the right of the

input word $w$. The two automata meet in the middle of $w$. $A_1$ checks whether the first half of the input consists solely of 0's and $A_2$ checks on its way back whether the second half of the input consists solely of 1's.

On the other hand it is known, that each non-regular language needs a recognition time of at least $n \log n$ for infinitely many $n$ on (1)–TM and therefore also on PTM with processor complexity 1 (see Theorem 5.2 below). ∎

**4.2 Lemma.** *Let $s$, $t$, $h$ and $h'$ be functions such that $h'$ is fully* PTM *processor constructible in space $s$, time $t$, and with $h'$ processors. Then it holds:*

$$\text{PTM–STP}(s,t,h) \subseteq \text{PTM–STP}(s, \mathrm{O}\left(\frac{st}{h'}\right), h') .$$

Observe that $h$ does not occur on the right hand side. In particular, this lemma says that, for PTM with "high" processor complexity, i.e. close to the space complexity, decreasing the number of processors does not result in such a big increase of the time complexity as it was the case in Lemma 4.1.

**Proof.** Let $P$ be a PTM with space, time, and processor complexities bounded by $s$, $t$ and $h$ resp., which is to be simulated by a PTM $P'$. For a given $P$-configuration $(p,b)$ tape square $i$ of $P'$ is used to hold the information $(p(i),b(i))$ of the corresponding $P$-cell $i$. In a first phase $P'$ positions $h'(n)$ finite automata on the tape (exploiting the fact that $h'$ is processor constructible). Then those $h'(n)-1$ at the left ends of the tape segments are synchronized (using the algorithm from Lemma 3.2 above) and simultaneously all of them start the sequential simulation of the behavior of the original PTM on its section.

The simulation on a segment can be carried out sequentially in a straightforward way: in a first sweep from left to right on each square $(p(i),b(i))$ is replaced by $(M',b') = \delta_P(p(i),b(i))$ (as described in Definition 2.1). On the way back the inscriptions of states of moving automata are cleared and written on the corresponding neighboring square. Since automata can move to the right as well as to the left, the sweep back has to be carried out in a zig-zag manner.

Automata working on the smaller segments always do a constant number of idle steps so that they spend exactly the same total amount of time for the simulation of one step of their segment as those automata working on the longer segments.

The segments are of length $\mathrm{O}(s/(h'-1))$ and therefore the simulation of one step of $P$ requires $\mathrm{O}(s/(h'-1))$ steps. ∎

## 4.2 A Tradeoff for a Specific Language

We shall now consider the language $L_{\mathrm{vv}} = \{v2^{|v|}v \mid v \in \{0,1\}^+\} \subseteq \{0,1,2\}^+$. First we describe a class of PTM recognizing $L_{\mathrm{vv}}$.

**4.3 Lemma.** *For every $a \in \mathbb{Q}$ with $0 < a < 1$ holds:*

$$L_{\mathrm{vv}} \in \text{PTM–STP}(n+1, \mathrm{O}\left(n^{2-a}\right), \mathrm{O}(n^a)).$$

**Proof.** A suitable PTM recognizing $L_{\mathrm{vv}}$ can work as follows: First it is checked that the input is of the form $\{0,1\}^*2^*\{0,1\}^*$ and that the three blocks are of equal length; this can be done using 3 finite automata simulating the signals of a standard CA technique [15]. Then $\lceil n^a \rceil$ finite automata are generated which do sweeps over the complete input word $w$. These automata mark the first unmarked symbol 0 or 1 (in the first third), save it in a register, and then move to the right, mark the first unmarked symbol 2 (in the second third), and finally mark the first not marked symbol 0 or 1 (in the third part) comparing it to the symbol stored in the register. If one of the comparisons fails, the word is rejected. If all comparisons succeed the input is accepted.

The difficult part is to show that for each $a \in \mathbb{Q}$ it is possible to generate $\lceil n^a \rceil$ finite automata. This requires some knowledge about efficiently PTM-computable and PTM-constructible functions. It can be shown that a sufficiently large class of functions satisfying

the necessary conditions contains all polynomials $n^k$ and is closed under composition and formation of inverse functions (e.g. $\lceil n^{1/k} \rceil$). The detailed constructions can be found in [19]. ∎

Next we will give a proof for a lower bound for $L_{\mathrm{vv}}$. It will use an idea similar to that of crossing sequences for Turing machines [6]. As a consequence one can deduce that the algorithm described in the previous proof is in fact quite good.

**4.4 Lemma.** *If $P$ is a* PTM *recognizing $L_{\mathrm{vv}}$, then* $\mathrm{Time}_P^2 \cdot \mathrm{Proc}_P \in \Omega\left(\frac{n^3}{\log^2 n}\right)$.

(Here we write $f \in \Omega(g)$ iff $\limsup_{n\to\infty} \frac{f(n)}{g(n)} > 0$.)

Before we actually prove this lemma, observe that together with Lemma 4.3 as an immediate consequence one gets the following corollary which will be exploited in the next section.

**4.5 Corollary.** *For rational numbers $0 < a < 1$ and $0 < \varepsilon < \frac{3}{2} - \frac{a}{2}$ holds:*

$$\text{PTM–STP}(n+1, \mathrm{O}\left(n^{3/2-a/2-\varepsilon}\right), \mathrm{O}(n^a)) \subsetneq \text{PTM–STP}(n+1, \mathrm{O}\left(n^{2-a}\right), \mathrm{O}(n^a))$$

**Proof of Lemma 4.4.** Let $P$ be a PTM recognizing $L_{\mathrm{vv}}$ with time complexity $t$ and processor complexity $h$. W.l.o.g. $t(n) \leq n^{3/2}$ (otherwise the lower bound is trivially satisfied). Let $A$ be the input alphabet of $P$ and $Q$ its set of states.

W.l.o.g. let $n$ be a multiple of 3. Consider an input word $w \in L_n := L_{\mathrm{vv}} \cap A^n$ of length $n$. For $1 \leq i \leq n-1$ let $c_w^{(i)} \in (2^Q \times 2^Q)^{t(n)}$ be the sequence of pairs of sets of states of the finite automata visiting the neighboring squares $i$ and $i+1$ during the computation for input $w$. The $c_w^{(i)}$ are called *crossing sequences*.

The number of crossing sequences with exactly $i$ pairs $(M_1, M_2) \neq (\emptyset, \emptyset)$ is $x_i = \binom{t(n)}{i} z^i$ where $z = |2^Q|^2 - 1$. Such pairs will be called *proper*. A crossing sequence is proper iff it contains at least one proper pair.

Similarly to [6] one can prove that there must not be two different words $w_1, w_2 \in L_n$ having a proper crossing sequence in common and there must not occur a proper crossing sequence at different positions for the same input word, i.e. $c_{w_1}^{(i_1)} \neq c_{w_2}^{(i_2)}$ unless $w_1 = w_2$ and $i_1 = i_2$. Otherwise one could split the space time diagrams between cells $i_1$ and $i_1 + 1$ and $i_2$ and $i_2 + 1$ resp., and glue together the left part of one diagram with the right part of the other diagram resulting in the space time diagram for an input $w$ which is accepted (because $w_1$ is accepted) although it is not in $L_{\mathrm{vv}}$ because either the first part of $w$ does not match the last part (if $w_1 \neq w_2$) or at least one part of $w$ doesn't have the correct length (if $i_1 \neq i_2$). Since $L_n$ contains $2^{n/3}$ words, there must be at least $(n-1)2^{n/3}$ different crossing sequences.

Hence the maximum number of proper pairs, occuring in at least one proper crossing sequence, must be at least $g(n)$, where $g(n)$ is determined by

$$\sum_{i=0}^{g(n)-1} x_i < (n-1)2^{n/3} \quad \text{and} \quad \sum_{i=0}^{g(n)} x_i \geq (n-1)2^{n/3}. \tag{1}$$

Consider the quantitiy

$$\sum_{w \in L_n} \sum_{j=1}^{n-1} S(c_w^{(j)})$$

where $S(c_w^{(j)})$ is the total number of occurrences of states in the crossing sequence $c_w^{(j)}$.

On one hand

$$\sum_{w \in L_n} \sum_{j=1}^{n-1} S(c_w^{(j)}) \leq 2 \cdot t(n) \cdot h(n) \cdot 2^{n/3} \tag{2}$$

since in every configuration occuring in any computation there are at most $h(n)$ automata and in the worst case each state in the time-space diagram is counted twice. On the other hand

$$\sum_{w \in L_n} \sum_{j=1}^{n-1} S(c_w^{(j)}) \geq \sum_{i=0}^{g(n)-1} x_i \cdot i \tag{3}$$

where the factors $i$ are due to the fact that there are $i$ pairs with at least one state in each of them for every crossing sequence with $i$ pairs. Hence from (2) and (3) we get[4]:

$$2t(n)h(n)2^{n/3} \quad \geq \quad \sum_{i=0}^{g(n)-1} x_i i \geq \left(\frac{g(n)}{2}\right) \cdot \left(\sum_{i=0}^{g(n)-1} x_i\right). \tag{4}$$

We now deduce lower bounds for both factors on the right using (1). First:

$$(n-1)2^{n/3} \leq \sum_{i=0}^{g(n)} x_i = \sum_{i=0}^{g(n)} \binom{t(n)}{i} z^i \leq \sum_{i=0}^{g(n)} (n^{3/2})^i z^i \leq (n^{3/2}z)^{g(n)+1}.$$

Hence there is a constant $c'$ such that for all sufficiently large $n$ holds:

$$g(n) \geq c' \frac{n}{\log n}. \tag{5}$$

A lower bound for $\sum_{i=0}^{g(n)-1} x_i$ can be obtained as follows:

$$\sum_{i=0}^{g(n)} x_i = \frac{z(t(n)+1-g(n))}{g(n)} x_{g(n)-1} + \sum_{i=0}^{g(n)-1} x_i \leq \frac{z(t(n)+1)}{g(n)} \sum_{i=0}^{g(n)-1} x_i.$$

Using (5) for all sufficiently large $n$ we get the following lower bound:

$$\sum_{i=0}^{g(n)-1} x_i \geq \frac{g(n)}{z(t(n)+1)} \sum_{i=0}^{g(n)} x_i \geq \frac{c'n(n-1)2^{n/3}}{2zt(n)\log n}.$$

Hence there is a constant $c''$ such that for all sufficiently large $n$ holds:

$$\sum_{i=0}^{g(n)-1} x_i \geq c'' \frac{n^2 2^{n/3}}{t(n)\log n} \tag{6}$$

Using (5) and (6) we can therefore continue (4):

$$2t(n)h(n)2^{n/3} \geq \left(\frac{g(n)}{2}\right) \cdot \left(\sum_{i=0}^{g(n)-1} x_i\right) \geq \frac{c'}{2} \frac{n}{\log n} c'' \frac{n^2 2^{n/3}}{t(n)\log n}$$

finally giving a constant $c$ such that for infinitely many sufficiently large $n$ holds:

$$t^2(n)h(n) \geq c \frac{n^3}{(\log n)^2}$$

■

**4.6 Open problem** *A comparison of the lower bound of Lemma 4.4 with the upper bound of* $\mathrm{Time}_P^2 \cdot \mathrm{Proc}_P = n^{4-a}$ *from Lemma 4.3 reveals a gap (which is very small if* $a$ *is close to 1). It is not known whether the upper or the lower bound or both for the recognition of* $L_{\mathrm{vv}}$ *can be improved.*

[4] using $x_i i + x_j j \geq \frac{i+j}{2}(x_i + x_j)$ for any increasing sequence $(x_i)_{i \in \mathbb{N}}$;

# 5 Comparison of PTM with CA and TM

In this section, the language recognition power of parallel Turing machines will be compared to that of one-dimensional cellular automata and to that of (sequential) one-head and multi-head Turing machines.

## 5.1 PTM versus CA

It has already been mentioned, that there is a close relation between PTM and CA. This becomes evident in the third part of the following theorem.

**5.1 Theorem.** *For all functions $s(n) \geq n$, $t(n) \geq n$, and $h(n) \geq 1$, where $h$ is fully* PTM *processor constructible in space $s$, time $t$, and with $h$ processors, holds:*

$$\begin{aligned} \text{PTM–STP}(\mathrm{O}(s), \mathrm{O}(t), \mathrm{O}(h)) &\subseteq \text{CA–ST}(\mathrm{O}(s), \mathrm{O}(t)) && (7)\\ \text{CA–ST}(\mathrm{O}(s), \mathrm{O}(t)) &\subseteq \text{PTM–STP}(\mathrm{O}(s), \mathrm{O}(st/h), \mathrm{O}(h)) && (8)\\ \text{PTM–STP}(\mathrm{O}(s), \mathrm{O}(t), \mathrm{O}(s)) &= \text{CA–ST}(\mathrm{O}(s), \mathrm{O}(t)) && (9) \end{aligned}$$

**Proof.**

**Inclusion (7):** A PTM $P = (Q_P, \ldots, B, \ldots)$ can be simulated by a CA $C$ with set of states $Q_C = 2^{Q_P} \times B$. A $P$-configuration $(p, b)$ in represented as a $C$-configuration in the obvious way: In its state $C$-cell $i$ stores the $P$-cell $(p(i), b(i))$. In its neighbors it observes $(p(i-1), b(i-1))$ and $(p(i+1), b(i+1))$. In a first step cell $i$ can determine $(M_i', b_i')$ (as described in Definition 2.1) from this information. In a second step it can use $(M_{i-1}', b_{i-1}')$ and $(M_{i+1}', b_{i+1}')$ to determine the new set of states it should store. Obviously, this is a local rule.

**Inclusion (8):** Because of Lemma 4.2 it suffices to prove that

$$\text{CA–ST}(\mathrm{O}(s), \mathrm{O}(t)) \subseteq \text{PTM–STP}(\mathrm{O}(s), \mathrm{O}(t), \mathrm{O}(s)).$$

Let $C$ be a CA with set of states $Q_C$ recognizing some language $L$. The basic idea is to simulate $C$ by a PTM $P$ using a chain of finite automata positioned on successive tape squares. Since they cannot "see" the states of automata on neighboring tape squares, the tape is used for the exchange of information.

Given some input word $w$ on its tape, in a first phase preceding the proper simulation, $P$ first generates finite automata on the tape squares $0, 1, \ldots, |w|, |w|+1$ storing the quiescent state, the symbols of $w$ and the quiescent state, and synchronizes them. Hence they start simultaneously to execute the following steps repeatedly:

- First each automaton moves one square to the left, writes its state on the tape, moves back to the right, reads the state of its right neighbor written on its square, stores it and deletes it from the tape by writing a special "erase symbol".
- Then each automaton does the same for the other direction.
- Now each finite automaton knows the state of "its own" cell and also the states of both neighboring cells and can simulate one state transition according to the rule of $C$.

A little bit of extra care has to be taken for the automata at both ends of the chain. W.l.o.g. consider the leftmost one. It can always find out that it is the leftmost one because there is never a left neighboring automaton replacing the erase symbol by a state of $C$ (the missing state has to be interpreted as the quiescent state of $C$). Whenever the leftmost automaton has to leave the quiescent state, it generates an additional automaton to be positioned on the left neighboring square in the quiescent state.

**Equality (9):** follows immediately from (7) and (8). ∎

In other words, PTM with an asymptotically maximal processor complexity are equivalent to cellular automata.

In the construction above the space complexity of the simulating machines may be larger by a constant of at most 2. Usually this can be compensated for by choosing a larger tape alphabet and/or set of states with one exception: In CA the cell holding the last input symbol can identify itself as such because it observes a cell in the quiescent state in its right neighbor. But in a PTM a finite automaton really has to visit the first blank tape square to find out that it has passed the last input symbol. Hence PTM with space complexity $n+1$ are equivalent to CA with space complexity $n$ (and also to those with space complexity $n+1$).

## 5.2 PTM versus (1)–TM

**5.2 Theorem.** *For all functions $s(n) \geq n$ and $t(n) \geq n$ holds:*

$$\text{PTM–STP}(\mathrm{O}(s)\,,\mathrm{O}(t)\,,1) = \text{(1)–TM–ST}(\mathrm{O}(s)\,,\mathrm{O}(t))$$

**Proof.** The constructions are straightforward. If a PTM has constant processor complexity 1, then in every configuration occuring in any computation for an input word there is exactly one finite automaton. Hence the transition function essentially degenerates to a function $\delta : S \times B \to S \times D \times B$, i.e. a (1)-Turing machine. The opposite simulation is trivial. ∎

**5.3 Theorem.** *For all functions $s(n) \geq n$, $t(n) \geq n$, and $h(n)$ holds:*

$$\text{PTM–STP}(\mathrm{O}(s)\,,\mathrm{O}(t)\,,\mathrm{O}(h)) \subseteq \text{(1)–TM–ST}(\mathrm{O}(s)\,,\mathrm{O}(st))$$

This follows from the fact that CA working in space $s$ and time $t$ can be simulated by (1)–TM in space $\mathrm{O}(s)$ and time $\mathrm{O}(st)$ (see [5]) and Theorem 5.1. For multi-head Turing machines the result can be improved as will be seen in the next subsection.

## 5.3 PTM versus TM

A time efficient simulation of Turing machines with several tapes and heads by PTM requires more processors than the simulation of (1)–TM. This is due to the fact that such Turing machines can communicate small amounts of information over a long distance *in one step*. This is impossible for PTM. As a compensation, they can communicate large amounts of information over a small distance in one step, if there is a large number of finite automata on the tape.

**5.4 Theorem.** *For all functions $s(n) \geq n$ and $t(n) \geq n$ holds:*

$$\text{TM–ST}(\mathrm{O}(s)\,,\mathrm{O}(t)) \subseteq \text{PTM–STP}(\mathrm{O}(s)\,,\mathrm{O}(t)\,,\mathrm{O}(s)) \tag{10}$$

$$\text{PTM–STP}(\mathrm{O}(s)\,,\mathrm{O}(t)\,,\mathrm{O}(h)) \subseteq \text{TM–ST}(\mathrm{O}(s)\,,\mathrm{O}\left(t\sqrt{sh}\right)) \tag{11}$$

**Proof.**

**Inclusion (10):** One can use the theorem of Stoß [12] for a linear time simulation of arbitrary TM by ones which have only one head per tape, and then simulate these by PTM as described by Wiedermann [16]. Independently the generalization of the latter construction for the case of several heads per tape had also already been described in [19].

**Inclusion (11):** Let $P$ be a PTM satisfying the resource bounds $s$, $t$ and $h$. The main idea for a TM $T$ simulating $P$ is as follows: On its tape $T$ maintains the description of a configuration of $P$ as in the proof of Lemma 4.2. But instead of making full sweeps over the whole tape segment simulating one step of $P$ at a time, it makes $q(n)$ sweeps

over relatively small "interesting" subsegments of the tape (simulating $q(n)$ steps of $P$ on it) while moving over large gaps without any finite automata to be simulated from one subsegment to another one only once every $q(n)$ simulation steps.

More precisely $T$ alternates between *partitioning phases* and *simulation phases*.

**Partitioning phase:** Denote by $S$ the smallest tape segment of $T$ comprising all used tape squares of $P$, let $s' = |S|$ denote the length of $S$ and $h'$ the number of finite automata positioned somewhere on $S$. In a partitioning phase $T$ first determines the value $q' = \sqrt{s'/h'}$ in unary as follows: First with one head $T$ makes one full sweep over $S$ while using a second head to count $h'$ in unary. Then $T$ makes a second full sweep over $S$ while simultaneously making sweeps over the $h'$. Each time the second head reaches the end of $h'$ a third head is moved one square. When finishing the sweep over $S$ the third head has moved $\lfloor s'/h' \rfloor$ tape squares. The square root of this number is determined by having two heads working on it, simulating the movement of the two signals used in the standard CA algorithm to mark the $\sqrt{k}$-th cell of $k$ cells.

Then $S$ is partitioned using $q'$: $T$ marks the left and right ends of all subsegments having the properties that

- they contain at least one non-empty $P$-cell,
- between two non-empty $P$-cells there are no more than $2q'-1$ empty $P$-cells, and
- to the left of the leftmost non-empty $P$-cell and to the right of the rightmost non-empty $P$-cells there are $q'$ empty $P$-cells.

**Simulation phase:** Note that because of the last condition $T$ can simulate $q'$ steps of $P$ on a subsegment without referring to any information outside the subsegment. In a simulation phase $T$ makes one pass over $S$. Whenever it enters a marked subsegment, it simulates $q'$ steps of $P$ on it. (It is no problem for $T$ to count to $q'$ because it has stored that value in unary.) Then $T$ leaves the subsegment and passes all empty $P$-cells until it enters the next marked subsegment.

After doing one partitioning and one simulation phase, $T$ has completely simulated $q'$ steps of $P$. The total time needed is at most $b_1 s' + b_2 h' q' q'$ for some constants $b_1, b_2$, where $h'q'$ is an upper bound on the total length of all marked subsegments which are passed $q'$ times. Therefore the average simulation time per step of $P$ is $b_1 s'/q' + b_2 h' q' = b_1\sqrt{s'h'} + b_2\sqrt{s'h'}$ which can obviously be bounded by $\mathrm{O}\left(\sqrt{sh}\right)$. ∎

At least in the case $t \in \Theta(n)$ (and hence $s \in \Theta(n)$) the simulation of multi-head TM by PTM is already quite processor efficient. $L_{\mathrm{vv}}$ can be recognized by a 3-head TM in linear time. According to Lemma 4.4 any PTM recognizing this language in linear time has to use $\Omega\left(\frac{n}{(\log n)^2}\right)$ processors.

Concerning the other inclusion it should be remembered that in the case $h \in \Theta(s)$ a (1)–TM was sufficient to achieve the same result (Theorem 5.3).

**5.5 Open problem** *While multi-head* TM *can be simulated by* PTM *in linear time, no such simulation is known for the reverse direction. In fact, one can suspect that there is none, because of the following informal observation. The first part of the above theorem can also be proved by giving a direct construction [19], in which almost all finite automata are used only for the "transport" of information, but not for the "processing" of information, i.e., it seems that in some sense not all the capabilities of* PTM *are needed in the construction.*

# 6 Complexity hierarchies by combinatorial arguments

The padding technique [11] can be used to prove the existence of arbitrarily large finite hierarchies of complexity classes for PTM.

**6.1 Definition.** *Let $f(n) \geq n$ be an increasing function, and $L \subseteq A^+$ a formal language with $\mathsf{3} \notin A$. We define*

$$L^f := \{v\mathsf{3}^{f(|v|)-|v|} \mid v \in L\}.$$

There is a rather close relation between the recognizability of a language $L$ and a padded version $L^f$, if the functions involved satisfy certain computability and/or constructibility requirements.

In what follows let $\bar{f} : \mathbb{N} \to \mathbb{N} : m \mapsto \min\{n \mid f(n) \geq m\}$ be the total function "similar" to $f^{-1}$ for any increasing $f : \mathbb{N} \to \mathbb{N}$. The functions occuring in the lemma below must be computable "sufficiently easy". For the sake of readability we'll write $g \diamond h$ for the function $(g \diamond h)(n) = \max(g(h(n)), n+1)$.

**6.2 Lemma.** *If $s$, $t$, $h$ and $f$ are increasing functions satisfying the conditions, that $f$ can be computed by a* PTM *within space $s \diamond \bar{f}$, time $t \diamond \bar{f}$ and with $h \diamond \bar{f}$ processors and that $n \log n \in \mathrm{O}\big(t \diamond \bar{f}\big)$ or $\log n \in \mathrm{O}\big(h \diamond \bar{f}\big)$, then the following propositions holds:*

$$L \in \text{PTM–STP}(s,t,h) \Longrightarrow L^f \in \text{PTM–STP}(s \diamond \bar{f}, \mathrm{O}\big(t \diamond \bar{f}\big), \mathrm{O}\big(h \diamond \bar{f}\big))$$
$$L^f \in \text{PTM–STP}(s,t,h) \Longrightarrow L \in \text{PTM–STP}(s \diamond f, \mathrm{O}\big(t \diamond f\big), \mathrm{O}\big(h \diamond f\big))$$

We omit the technical but straightforward proof.

The upper and lower bounds on the product $\mathrm{Time}_P^2 \mathrm{Proc}_P$ for the recognition of $L_{\mathrm{vv}}$ are rather close to each other if $\mathrm{Time}_P$ is sufficiently close to $n$. Because of the result above, this fact is passed on to the languages $L_{\mathrm{vv}}^f$, if, for example, $f = n^c$ where $c > 1$ is a suitable rational number.

**6.3 Theorem.** *For all $b, b_2 \in \mathbb{Q}$ with $0 < b < b_2 < 1$ and $b_2 < \frac{2+b}{3}$ there is a $b_1 \in \mathbb{Q}$ with $b < b_2 < b_1 < 1$ such that it holds:*

$$\text{PTM–STP}(n+1, \mathrm{O}\big(n^{2-b_1}\big), \mathrm{O}\big(n^b\big)) \subsetneq \text{PTM–STP}(n+1, \mathrm{O}\big(n^{2-b_2}\big), \mathrm{O}\big(n^b\big))$$

As can be seen from the following proof it is possible to make the difference $b_1 - b_2$ arbitrarily small by choosing a sufficiently large $b$.

**Proof.** For given $b, b_2$ the condition $b_2 < \frac{2+b}{3}$ is equivalent to $\frac{1}{2} + \frac{b_2}{2} + \frac{b_2 - b}{4} < 1$. Hence one can choose a $b_1$ where $1 > b_1 > \frac{1}{2} + \frac{b_2}{2} + \frac{b_2-b}{4}$. Furthermore $0 < b < b_2 < 1$ implies that as a consequence $b_2 < b_1$. Consider $c = \frac{2}{2-b_2+b}$ now. We claim that the language $L_{\mathrm{vv}}^{n^c}$ is a witness for the properness of the inclusion: First it should be noted, that $c > 1$, $f(n) = n^c$ satisfies the conditions of Lemma 6.2, and $\bar{f} = n^{1/c}$.

Since $c(2-b_2)+cb = 2$, it follows from Lemma 4.3 that $L_{\mathrm{vv}} \in \text{PTM–STP}(n+1, \mathrm{O}\big(n^{c(2-b_2)}\big), \mathrm{O}\big(n^{cb}\big))$. Hence $L_{\mathrm{vv}}^{n^c} \in \text{PTM–STP}(n+1, \mathrm{O}\big(n^{2-b_2}\big), \mathrm{O}\big(n^b\big))$ because of Lemma 6.2.

On the other hand a straightforward computation shows that $2c(2-b_1) + cb < 3$, i.e., $(n^{c(2-b_1)})^2 n^{cb} \in \mathrm{O}\big(n^{3-\varepsilon}\big)$. Therefore Lemma 4.4 assures that $L_{\mathrm{vv}} \notin \text{PTM–STP}(n+1, \mathrm{O}\big(n^{c(2-b_1)}\big), \mathrm{O}\big(n^{cb}\big))$ and an application of Lemma 6.2 shows $L_{\mathrm{vv}}^{n^c} \notin \text{PTM–STP}(n+1, \mathrm{O}\big(n^{2-b_1}\big), \mathrm{O}\big(n^b\big))$. ∎

This means that for *fixed* space complexity of $n+1$ and for some *fixed* processor complexities there are arbitrarily large finite hierarchies of time complexity classes. Similarly one can prove that for *fixed* space complexity of $n+1$ and for some *fixed* time complexities there are arbitrarily large finite hierarchies of processor complexity classes.

Note that these hierarchies all completely lie within e.g. $\text{PTM–STP}(n+1, n^2, \mathrm{O}(n)) \subseteq \text{CA–ST}(n, n^2)$. Hence we have found some structure in time complexity classes for cellular automata with a linear space bound. Furthermore these hierarchies are due to a *parallel* model, namely PTM with processor complexity $n^b$ for any $0 < b < 1$. Choosing $b$ large results in a model which is in some sense close to CA, but not quite.

**6.4 Open problem** *It is not known whether Theorem 6.3 also holds for* PTM *with maximum processor complexity, i.e. cellular automata.*

Analogously to Theorem 6.3 one can also derive a processor complexity hierarchy in the case of fixed time complexity:

**6.5 Theorem.** *For all $b, b_2 \in \mathbb{Q}$ with $0 < b_2 < b < 1$ and $b_2 < \frac{2-b}{3}$ there is a $b_1 \in \mathbb{Q}$ with $0 < b_1 < b_2$ such that it holds:*

$$\text{PTM–STP}(n+1, \mathrm{O}(n^{2-b}), \mathrm{O}(n^{b_1})) \subsetneq \text{PTM–STP}(n+1, \mathrm{O}(n^{2-b}), \mathrm{O}(n^{b_2}))$$

**Proof.** One has to choose a $b_1 < \frac{b}{2} + \frac{3b_2}{2} - 1$ and $c = \frac{2}{2-b+b_2}$. ∎

# 7 Diagonalization I: fixed processor complexity

For the rest of this paper let $A$ be an arbitrary but fixed input alphabet with at least two symbols.

In this section we will prove:

**7.1 Theorem.** *Let $s$ and $t$ be two functions such that $s$ is fully* PTM *space constructible in time $t$ and $t$ is* PTM *computable in space $s$ and time $t$ and let $h \geq \log$. Then:*

$$\bigcup_{\gamma \notin \mathrm{O}(1)} \text{PTM–STP}(\Theta(s/\gamma), \Theta(t/\gamma), \mathrm{O}(h)) \subsetneq \text{PTM–STP}(\mathrm{O}(s), \mathrm{O}(t), \mathrm{O}(h)).$$

First we will give the definition of PTM computability. Then it will be shown how PTM configurations can be encoded in such a way that it is not too difficult to describe a universal PTM $U$. Finally $U$ is used in the diagonalization proof of Theorem 7.1.

Let $\mathrm{bin}(x) \in \{\mathtt{0},\mathtt{1}\}^+$ denote the usual binary representation of a natural number $x$ without leading zeroes (except for $x = 0$) and for $k \geq |\mathrm{bin}(x)|$ let $\mathrm{bin}_k(x) := \mathtt{0}^{k-|\mathrm{bin}(x)|}\,\mathrm{bin}(x)$ be the binary representation of $x$ using $k$ bits.

A function $f$ is called PTM *computable* in space $s$, time $t$, and with $h$ processors iff there is a PTM $P = (Q, \ldots)$ with the following properties:

- $Q$ contains the two states $\mathtt{0}, \mathtt{1}$.
- For $x \in \mathbb{N}_+$ let $c_x$ denote a configuration where for each $i$ with $1 \leq i \leq |\mathrm{bin}(x)|$ on tape square $i$ is a finite automaton in state $x_i$ such that $x_1 \cdots x_k$ is $\mathrm{bin}(x)$. Then for each $x \in \mathbb{N}_+$ starting from $c_x$ $P$ must reach configuration $c_{f(x)}$.
- For all $x \in \mathbb{N}_+$ in order to compute $c_{f(x)}$ from $c_x$ $P$ needs at most $s(x)$ tape squares, at most $t(x)$ steps, and at most $h(x)$ finite automata.

It should be noted, that the resource bounds are formulated in terms of $x$ and *not* in terms of $|\mathrm{bin}(x)|$. This is in accordance with the fact, that the complexity measures are defined in terms of input length of which an input word can be considered as a *unary* representation.

The above notion (as well as that of constructibility in Subsection 3.2) has been defined in such a way that in all cases it is meaningful to require in addition that the tape is not used during the computations. This will be used in Section 8.

The first step towards a proof of Theorem 7.1 is the description of a coding of PTM. A PTM $P = (Q, q_0, F_+, F_-, B, A, \Box, \delta)$ will be described as a word $\mathrm{cod}(P) \in C^*$ over the *coding alphabet*[5] $C = \{\mathtt{[},\mathtt{]},\mathtt{0},\mathtt{1}\}$. cod will be chosen such, that it can be easily checked whether a word $w \in C^*$ is a coding of a PTM, and if it is that it can be used very easily for the efficient simulation of the encoded PTM.

From now on we will always assume that the input alphabet is totally ordered and contains all symbols of $C$ as its first symbols in some fixed order. Furthermore we assume that the set of states and the tape alphabet of each PTM are totally ordered in such a way that w.l.o.g. the initial state is the first in the enumeration of $Q$ and in the enumeration of $B$

[5] $C$ is chosen to be convenient; of course two symbols are sufficient.

the blank symbol is the first, followed by all input symbols. Such PTM will be said to be in *normal form*.

Let $P$ be an arbitrary PTM in normal form and $k = \max\{|Q|, |B|\}$. Sets of states and tape symbols are encoded as $k$-bit strings as follows: $R \subseteq Q$ is encoded as $\mathrm{cod}_q(R) = \mathtt{[}x_0x_1\cdots x_{k-1}\mathtt{]} \in \mathtt{[\{0,1\}^k]}$ with $x_i = \mathtt{1}$ iff $i < |Q| \wedge q_i \in R$, and analogously for tape symbols. Let $\mathrm{cod}_b(b_i)$ denote the coding of $\{b_i\}$.

For a set $M' \subseteq Q \times D$ let $M'[d] = \{q \mid (q,d) \in M'\}$. A single "entry" $\delta(R,b) = (M',b')$ of the local transition function is encoded as the word $\mathrm{cod}_r(R,b,M',b') =$

$$\mathtt{[}\,\mathrm{cod}_q(R)\,\mathrm{cod}_b(b)\,\mathrm{cod}_q(M'[-1])\,\mathrm{cod}_q(M'[0])\,\mathrm{cod}_q(M'[1])\,\mathrm{cod}_b(b')\mathtt{]}\,.$$

The coding $\mathrm{cod}_\delta(\delta)$ of a complete transition function $\delta$ is the concatenation of all codings $\mathrm{cod}_r(\cdots)$ of entries in lexicographical order.

The *coding* of a PTM $P = (Q, q_0, F_+, F_-, B, A, \Box, \delta)$ is the word $\mathrm{cod}(P) =$

$$\mathtt{[}\,\mathrm{cod}_q(Q)\,\mathrm{cod}_q(F_+)\,\mathrm{cod}_q(F_-)\,\mathrm{cod}_b(B)\,\mathrm{cod}_\delta(\delta)\mathtt{]}\,.$$

Let $z = |Q|$, $y = |B|$ and hence $k = \max\{z, y\}$. Obviously $|\mathrm{cod}_\delta(\delta)|$ is the dominating summand for the length $l = |\mathrm{cod}(P)|$. $|\mathrm{cod}_\delta(\delta)|$ is proportional to $(2^z y)k$ and hence $k \leq d_1\sqrt{l}$ for some constant $d_1$ always holds.

Of course there is a PTM which can check whether a word $w \in C^*$ is the coding of a PTM. Define $L_{\mathrm{cod}} = \{cod(P) \mid P \text{ is a PTM}\}$. Membership in $L_{\mathrm{cod}}$ can be checked rather efficiently:

**7.2 Lemma.** *There is a* PTM *recognizing the language of* PTM *codings* $L_{\mathrm{cod}}$ *in space* $n+1$, *time* $\mathrm{O}(n)$ *and with* $\mathrm{O}(\log n)$ *processors without ever writing something on the tape.*

**Proof.** First an increasing chain of successive finite automata is used as a binary counter to determine the length of the input $w$. The resulting block of $\Theta(\log n)$ automata can then be used to check all the syntactic requirements given above by sweeping over $w$ a finite number of times. ∎

We are now ready to describe how an arbitrary configuration $c$ of an arbitrary PTM $P$ can be encoded in such a way as a configuration $\mathrm{cod}(c)$, that it will be possible to describe an efficient universal simulator $U$ afterwards which simulates the step $c \mapsto \Delta_P(c)$ (by computing $\mathrm{cod}(\Delta_P(c))$).

Let $P$ be a PTM in normal form and $c$ a configuration of $P$ in which only a finite number of tape squares is used. $c$ will be encoded as the inscription on a tape with seven tracks of some PTM $U$ (see Figure 1). The inscription is divided into a finite number of successive finite *segments* of equal length. All other squares contain the $\Box$ symbol (of $U$). All segments have the same length and structure, which will be described now.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| track 1: | ] | [ | | | ] | [ | |
| track 2: | ··· | [01100] | [00000] | ··· | [11101] | ··· | |
| track 3: | ··· | [01000] | [10000] | ··· | [00001] | ··· | |
| track 4: | | | | | | | |
| track 5: | | | | | | | |
| track 6: | $w_{\mathrm{suf}}$ | $\mathrm{cod}(P)$ | | | $w_{\mathrm{suf}}$ | $\mathrm{cod}(P)$ | |
| track 7: | | | | | | | |

Figure 1: The coding of a PTM on the tape of a universal simulator PTM. The vertical lines separate cell blocks and the vertical double lines segments.

- Each segment encodes the inscriptions of a successive number of tape squares $l, \ldots, r$ of $c = (p, b)$ and the states of the finite automata visiting them.
- On track 1 the leftmost and rightmost tape square of a segment are marked with a [ and a ] respectively. The other squares are empty.

- Track 2 contains the concatenation $\mathrm{cod}_q(p(l)) \cdots \mathrm{cod}_q(p(r))$ of the codings of the states.
- Track 3 contains the concatenation $\mathrm{cod}_b(b(l)) \cdots \mathrm{cod}_b(b(r))$ of the codings of the square symbols. (Remember that codings of state sets and symbols have the same length $k+2$.)

  Let us call $k+2$ successive tape squares which contain the coding of a set of states and of a tape symbol a *cell block*. Hence each segment consists of a number of cell blocks.
- The fourth and fifth track will be used by $U$ only during the simulation.
- Track 6 contains a word of the form $\mathrm{cod}(P)w_{\mathrm{suf}}$ with $w_{\mathrm{suf}} \in \{\texttt{[]}\}^*\{\lambda, \texttt{]}\}$. ($\lambda$ denotes the empty word.) The length of a segment has to be an integral multiple of $k+2$. Since this may not be the case for $\mathrm{cod}(P)$ we allow padding it with $w_{\mathrm{suf}}$ but require that the length of the segment is such that the length of $w_{\mathrm{suf}}$ is less than $k+2$.
- The seventh track will be used by $U$ only during the simulation.
- On the leftmost tape square of the segment there is a finite automaton $S$ in some distinguished state $s$.

We call a tape inscription a *coding* $\mathrm{cod}(c)$ *of a configuration* if the non-blank part of the tape consists of a finite number of segments (as described above) where the leftmost and the rightmost one on the second and third track only contain $\mathrm{cod}_q(\emptyset)$ and $\mathrm{cod}_b(\Box)$ and the segments encompass all used squares of $c$. The length of the coding of a configuration is the number of tape squares used by all tape segments together.

**7.3 Lemma.** *There is a* universal simulator PTM $U$ *with the following properties: For each* PTM $P$ *with* $l = |\mathrm{cod}(P)|$ *and each configuration* $c$ *of* $P$ *given a coding* $\mathrm{cod}(c)$ $U$ *computes a coding* $\mathrm{cod}(\Delta_P(c))$ *in a time proportional to* $l^2$. *The number of finite automata needed is at most proportional to the number of automata occuring in* $c$ *or* $\mathrm{cod}(c)$. *If* $\mathrm{cod}(c)$ *is chosen as short as possible*[6], *then the space complexity of* $U$ *is at most* $dl$ *times bigger than that of* $P$ *(for some constant* $d$*).*

**Proof.** Let $P$ be an arbitrary PTM with $u = \mathrm{cod}(P)$, $l = |u|$ and $k$ as above.

The simulation of one step of $P$ consists of 3 phases. First we describe a simplified version which does not satisfy the processor bound. Instead during all phases there will be exactly one finite automaton working on each segment.

For the following note that $S$ can easily count to $k$, for example using a marker on track 1, since the length of cell blocks is $k+2$.

1. To simplify the simulation in the first phase track 7 is used to generate a "compacted" and easier to use description of the local transition function to be used. For the compacted form imagine the track divided into 8 subtracks, which are used to hold on top of each other the following informations in one cell block. For some entry $\delta(R, b) = (M', b')$ these are $\mathrm{cod}_q(R)$, $\mathrm{cod}_b(b)$, $\mathrm{cod}_q(M'[-1])$, $\mathrm{cod}_q(M'[0])$, $\mathrm{cod}_q(M'[1])$, and $\mathrm{cod}_b(b')$ and furthermore $\mathrm{cod}_q(F_+)$ and $\mathrm{cod}_q(F_-)$.

   The inscription of track 7 can be generated by $8(k+2)$ sweeps over the whole segment.

2. Then each finite automaton $S$ simulates one step of $P$ on its segment. In order to do that, the codings on tracks $2, \ldots, 5$ of a cell block are compared to all "entries" as they can be found on track 7 of the cell blocks of the segment. For the matching entry the transition is simulated.

   This can be realized by shifting track 7 "cyclically" through all tape squares of the segment. Additionally every $k+2$ steps the finite automaton checks whether a matching rule has reached a cell block. If it has, the coding $\mathrm{cod}_b(b')$ of the new tape symbol is copied to track 3, $\mathrm{cod}_q(R[0])$ is copied to track 2, and $\mathrm{cod}_q(R[-1])$ and $\mathrm{cod}_q(R[1])$

[6] Given a coding $\mathrm{cod}(c)$ one can construct longer ones by adding segments corresponding to empty $\Box$ squares on either side.

to tracks 4 and 5. On track 1 a mark is written indicating that for the current cell block one step has already been simulated in the current phase. Before simulating a transition step in a cell block the finite automaton first checks whether there is already such a mark, in which case it does not change the tape contents.

3. If the rotation of track 7 is finished, i.e. if each entry has been compared to each cell block, corresponding to the movement of the finite automata in the simulated PTM the contents of track 4 and 5 have to be shifted one cell block to the left and to the right respectively, and the information about the states has to be added to track 2.

It should be noted that during the last two phases a finite automaton has to leave its own segment and move $k+2$ squares into its neighboring segments.

The length of a segment is smaller than $l+k<2l$. The space complexity of the above algorithm is determined by the maximum number $m$ of cells needed in $c$ or $\Delta_P(c)$. If a short coding of $c$ is used, $4+m/l$ is an upper bound for the number of segments. Hence $U$ needs at most $d_1lm$ tape squares (for some constant $d_1$).

Adding the time complexities for the 3 phases gives an upper bound of $d_2kl+d_3l^2+d_4kl\leq d_5l^2$ for some constants $d_i$.

The description above assumed one finite automaton on each segment. Hence the processor complexity would be $d_6m/l$ which can be much larger than the number of automata occuring in $c$ or $\Delta_P(c)$. But this only happens if the work of the PTM is really simulated, even in segments where "nothing happens" because there are currently no finite automata to be simulated. To avoid this overhead, the above procedure can be modified. Assume that when the simulation starts, there are only finite automata on segments where there really is something to be simulated. This condition can be made an invariant by adding a fourth phase which ensures it also at the end of the simulation of one step:

4. Let each finite automaton make a complete sweep over the two segments which are adjacent to its own, checking whether something needs to be simulated in the next step but currently there is no finite automaton for this segment. If this is true, a finite automaton for the segment is generated. Of course one has to take care that the newly generated automata afterwards start working synchronously with the already existing ones. Note that although at first glance this looks like the problem mentioned in the sketch of proof of Lemma 4.2 it is not. In fact an FSSP would need a time proportional to the space complexity which would be much more for the simulation of one step than we would like to spend. Instead one can use the fact that neighboring (indeed all) segments have equal length $r$ and use a simple three signal construction as one often encounters in CA algorithms. Assume that at the left end of a segment is a finite automaton $S_1$ and that on the neighboring one to the right an automaton $S_2$ has to be generated. To this end $S_1$ moves to the right end of its segment and then back to the left with an average speed of $2/3$ (cells per step) returning after $3r$ steps. When $S_1$ begins to move to the right two other automata $H_1$ and $H_2$ are started at the same square. $H_1$ moves with speed 1 crosses two segments and then begins to return. $H_2$ moves with speed $1/3$. Hence the two meet in the middle of $2r$ cells, i.e. at the beginning of the neighboring segment, where they melt together to $S_2$ after $3r$ steps, too.

   Also a finite automaton which finds during the sweep in the fourth phase, that it has nothing to simulate on its segment in the next step, disappears. In this way the number of finite automata needed for the simulation can be reduced to at most a constant times the number of finite automata occuring in $c$ or $\Delta_P(c)$. ∎

From Lemmata 7.3 and 7.2 immediately follows:

**7.4 Corollary.** *There is a* PTM $U$ *recognizing the language* $\{uv \mid u\in L_{\mathrm{cod}}\wedge v\in L(\mathrm{cod}^{-1}(u))\}$ *such that for all* $u\in L_{\mathrm{cod}}$, $P=\mathrm{cod}^{-1}(u)$, *and all* $v$ *holds:*

- $\mathrm{space}_U(uv)\in \mathrm{O}(|u|\cdot \mathrm{space}_P(v))$,

- $\mathrm{time}_U(uv) \in \mathrm{O}\big(|u|^2 \cdot \mathrm{time}_P(v)\big)$, *and*
- $\mathrm{proc}_U(uv) \in \mathrm{O}(\max\{\log|u|, \mathrm{proc}_P(v)\})$,

*and for words* $w \notin L_{\mathrm{cod}} \cdot A^*$ *holds*

- $\mathrm{space}_U(w) \in \mathrm{O}(|w|)$,
- $\mathrm{time}_U(w) \in \mathrm{O}(|w|)$ *and*
- $\mathrm{proc}_U(w) \in \mathrm{O}(\log|w|)$.

We can now prove the main theorem of this section.

**Proof of Theorem 7.1.** Let $s$ and $t$ be two functions such that $s$ is fully PTM space constructible in time $t$ and $t$ is PTM computable in space $s$ and time $t$. We will describe a PTM $D$ recognizing a language in space $\mathrm{O}(s)$ and in time $\mathrm{O}(t)$ and prove that it is not in PTM–ST$(\Theta(s/\gamma), \Theta(t/\gamma))$ for any $\gamma \notin \mathrm{O}(1)$.

An input word $w \in C^*$ is processed in four phases:

1. $D$ checks whether $w = uv$ for a syntactically correct coding $u$ of a PTM $P$ and an arbitrary suffix $v$. Because of the way we have defined encodings, for each $w$ there is at most one prefix from $L_{\mathrm{cod}}$.
2. In the following let the tape be divided into 2 tracks. On track 1 $D$ computes $t(|w|)$ and stores the result in a chain of subsequent automata which will later act as a counter. Furthermore $D$ marks a section of $s(|w|)$ tape squares by two automata at the ends. In the sequel $D$ always rejects $w$ whenever it would have to use a square outside the marked area.
3. Next, $D$ tries to generate a shortest coding of $c_w$, thought of as a configuration of $P$ (if there is sufficient space), and initializes the second track, consisting of seven subtracks, as it is needed for the universal simulator as described in the proof of Lemma 7.3.
4. Finally on the second track $D$ works as the universal simulator. Simultaneously in each step of $D$ the counter built up from finite automata in phase 2 is decremented by 1. $D$ stops the simulation if either the counter has reached 0 or $P$ had reached a final configuration. If $P$ would accept $w$ then $D$ rejects it. If $P$ would reject $w$ or if the simulation had to be stopped prematurely then $D$ accepts $w$.

Obviously $D$ satisfies the space and time requirements $s$ and $t$.

Now assume that there were a PTM $P$ recognizing $L(D)$ and working in space $\Theta(s/\gamma)$ and time $\Theta(t/\gamma)$ for some $\gamma \notin \mathrm{O}(1)$. To deduce a contradiction let $u = \mathrm{cod}(P)$ and observe first that according to Lemma 7.3 there are constants such that for all $v \in A^+$ the space and time needed by the universal simulator for the input $uv$ can be bounded from above by

$$\begin{aligned} d_1|u| \cdot \mathrm{space}_P(uv) \quad &\text{which is} \quad \leq d_1|u|\frac{s(|uv|)}{\gamma(|uv|)} \text{ and} \\ d_3|u|^2 \cdot \mathrm{time}_P(uv) \quad &\text{which is} \quad \leq d_4|u|^2 \cdot \frac{t(|uv|)}{\gamma(|uv|)} \end{aligned}$$

Since $\gamma$ goes to infinity on a subset of $\mathbb{N}_+$ there is a $v'$ satisfying $s(|uv'|) \geq d_1|u| \cdot \mathrm{space}_P(uv')$ and $t(|uv'|) \geq d_3|u|^2 \cdot \mathrm{time}_P(uv')$. Hence for the input $uv'$ $D$ can simulate all steps of $P$ for the same input until it reaches a final configuration. Therefore it is not only the case that if $D$ rejects $uv'$ this is because $P$ would accept it, but also if $D$ accepts $uv'$ this is because $P$ would reject it. Hence $L(D)$ and $L(P)$ differ by $uv'$ contrary to what we had assumed. ∎

Using Theorem 5.1 as an easy corollary one obtains the following results for cellular automata.

**7.5 Corollary.** *Let $s$ and $t$ be two functions such that $s$ is fully* PTM *space constructible in time $t$ and $t$ is* PTM *computable in space $s$ and time $t$. Then:*

$$\bigcup_{\gamma \notin O(1)} \text{CA–ST}(\Theta(s/\gamma), \Theta(t/\gamma)) \subsetneq \text{CA–ST}(O(s), O(t))$$

$$\text{CA–ST}(o(s), o(t)) \subsetneq \text{CA–ST}(O(s), O(t))$$

$$\text{CA–T}(o(t)) \subsetneq \text{CA–T}(O(t))$$

The second and third inequation are simply special cases of the first one. These results provide smaller gaps than theorems 6 and 7 in the article by Ibarra, Kim and Moran [8]. (The proof of their first result is incomplete since it applies a theorem to Turing machines with one tape and one head although it has been proved only for Turing machines with at least two tapes by Paul [10].)

# 8 Diagonalization II: fixed space complexity

While the results in the previous section are interesting on their own, they do *not* solve the open problem for cellular automata with space complexity $s(n) = n$ mentioned in the introduction. There are two reasons for the increase of the space complexity in the above constructions. The universal simulator has to cope with all PTM having arbitrarily large state sets and tape alphabets which have to be encoded using one fixed state set and one fixed tape alphabet. Hence the coding of a subset of states or of a tape symbol may become arbitrarily long resulting in a space complexity for the universal simulator which cannot be bounded by a constant times the space complexity of the simulated PTM.

Two possibilities come to mind how this problem might be circumvented. The first is bounding the size of sets to be encoded. Of course it is not possible to fix the sizes of both the set of states and the tape alphabet, since this would mean to consider only a finite number of PTM. But we will fix the size of the tape alphabet. At least this does not cut down the number of languages recognizable within some space and time bounds $s$ and $t$, because one can always increase the set of states and/or the processor complexity in order to be able to store enough information.

The other possibility, which will be used for the states, is using a more efficient coding. If for example during the computations of a PTM most of the tape squares are empty, then it would be preferable to encode the empty set (of states) by a much shorter word than other subsets. In the construction below the following version of this idea will be employed: For each tape square one bit is used to distinguish between empty and non-empty ones. And only for non-empty ones the set of states will be stored similar to the form in the previous section.

There all of the constructibility and computability notions have been defined in such a way that the tape inscription at the end of a computation is the same as at the beginning. One can therefore define corresponding notions with the additional requirement that no tape square is written during the computations. These are used in the main theorem of this section:

**8.1 Theorem.** *Let $s$, $t$ and $h$ be three functions such that $s$ is fully* PTM *space constructible in time $t$ and with $h$ processors, and that $t$ and $h$ are* PTM *computable in space $s$ and time $t$ and with $h$ processors such that in all cases the tape is not written. Then:*

$$\bigcup_{\gamma \notin O(1)} \text{PTM–STPA}(s, \Theta(t/\gamma), \Theta(h/\gamma), b) \subsetneq \text{PTM–STPA}(s, \Theta(st), O(h), b).$$

Here we use the extended notation PTM–STPA$(s, t, h, b)$ to indicate the cardinality $b$ of the tape alphabet, too. I.e. in this theorem it is assumed that only tape alphabets of a fixed size are used, whereas in the previous section arbitrary tape alphabets were allowed.

For the proof we proceed analogously to the previous section. First a new coding of PTM configurations is presented. Then a universal simulator working with these codings is described, which is finally used in the diagonalization proof of Theorem 8.1.

From now on without loss of generality let $B$ be a *fixed* tape alphabet with $C = \{\mathtt{0}, \mathtt{1}, \mathtt{[}, \mathtt{]}\} \subseteq B$. (Again we only use 4 symbols because it is more convenient; 2 symbols would be enough.)

A configuration $c'$ of a PTM $U$ is called a coding of a configuration $c = (p, b)$ of a PTM $P$ if the following holds:

- The tape inscriptions of $c$ and $c'$ are identical.
- On each tape square which is non-empty in $c$ there is an automaton in $c'$ in a designated state $\bullet$. Such automata are called *proper marking automata*. On tape squares which do not have a proper marking automaton but which are immediately neighbored to such a square there is an *improper marking automaton* in state $\circ$.
- A section of tape squares of maximal length with the property that on each of them there is a (proper or improper) marking automaton but there are no two improper marking automaton on neighboring squares is called a *state section*.
- On some square to the left of the leftmost improper marking automaton there is a *border automaton* in state `[` and on some square to the right of the rightmost improper marking automaton there is a border automaton in state `]`.
- Starting at the left border automaton there is a chain of *coding automata* which ends on some square to the left of the right border automaton. Each of the coding automata consists of 7 registers. The resulting 7 chains of registers play similar roles as the tracks on the tape in the coding used in the previous section, and are henceforth called tracks again
  - On track 2 codings $\mathrm{cod}_q(R)$ of the sets of states are stored. If $j$ is the number of the tape square with the $i$-th ($i \geq 1$) proper marking automaton, then the coding automata with numbers $(i-1)(k+2)+1, \ldots, (i-1)(k+2)+k+2$ store $\mathrm{cod}_q(p(j))$. A chain of $k+2$ coding automata storing a coding $\mathrm{cod}_q(R)$ are called a cell block again.
  - On track 1 beginning and end of the codings of each state section are marked.
  - Track 3 is empty but it will be used during the simulation for storing the codings of marked tape squares.
  - Tracks 4, 5, 6 and 7 are used for the same purposes as in Section 7: On them are stored shifted codings of sets of states, $\mathrm{cod}(P)$ in standard form and in the "compacted" form.

**8.2 Lemma.** *There is a universal simulator* PTM $U$ *with the following properties: For each* PTM $P$ *with* $l = |\mathrm{cod}(P)|$ *and each configuration* $c$ *of* $P$ *given a coding* $\mathrm{cod}(c)$ *of length* $S(c)$ $U$ *computes a coding* $\mathrm{cod}(\Delta_P(c))$ *in a time at most* $d_1 S(c)$. *If* $H(c)$ *denotes the number of processors occuring in* $c$, *the simulation needs a space of at most* $d_2 \max\{S(c), \sqrt{l}\,H(c)\}$ *cells and at most* $d_3\sqrt{l}\,H(c)$ *processors (for some constants* $d_1, d_2, d_3$*).*

**Proof.** Let $P$ be an arbitrary PTM with $u = \mathrm{cod}(P)$, $l = |u|$ and $k = \max\{|Q|, |B|\}$. The simulation of one step of $P$ consists of 5 phases.

1. During the first phase in each cell block of coding automata the coding of the symbol on the tape square of the corresponding marking automaton has to be generated. To this end a signal automaton moves from the left to the right border automaton with speed 1/3. Whenever it arrives at a tape square with a marking automaton, it generates yet another one carrying the read symbol to the left until it meets the first coding automaton with an empty third register. In it and the neighboring third registers to the left, the coding of the symbol is stored.

2. In the next phase analogously to the description in the proof of Lemma 7.3 the codings of the new sets of state and the new tape symbol are generated in the coding automata.

3. Afterwards the real tape inscription has to be changed according to the just computed codings. At the left border automaton two automata are started. One moves to the first marking automaton. Its task is to indicate always the tape square which has to be updated next. The other automaton moves to the right with speed 1/3. Whenever it reaches the left end of the coding of symbol, it starts an automaton which reads the coding, moves to the right (with speed 1) to the automaton indicating the square to be updated, updates it and vanishes.

   When the second automaton reaches the right border automaton, the new tape inscription is correct and the next phase is started.

   The remaining two phases are needed because during the simulation of one step it may happen that two state sections are melting to one and/or that a state section splits into two. Hence the number, types and positions of the marking automata have to be changed (phase 4) and the states of the coding automata have to be changed accordingly (phase 5).

4. In phase 4 each marking automaton receives together with the new tape symbol the information, whether it will be a proper or an improper one, and it assumes the corresponding state. After this has been done it may be necessary to delete and/or generate improper marking automata such that again each proper marking automaton has two neighboring marking automata and each improper marking automaton has at least one proper neighboring marking automata. This can be done by an automaton $G$ moving from e.g. the right border automaton to the left one by doing three steps for each square: looking ahead to the next square, coming back to the current one (updating the marking automaton if necessary) and moving forth again to the next square.

5. Finally the states of the coding automata must be adapted to the new positions of the marking automata. In fact, this "phase" is interleaved with the previous one. Whenever the automaton $G$ generates, meets, changes or deletes a marking automaton during phase 4 it sends an automaton to the left with this information and (if appropriate) with the information whether two state sections have become one or one has become two. Since the marking automata are visited from right to left and the corresponding coding automata are positioned in the same order, the cell block where the information has to be processed can again be indicated by a finite automaton.

   The most difficult case is the generation of an additional improper marking automaton and the insertion of the corresponding additional cell block of coding automata between already existing ones. Of course the latter cannot be shifted to the right immediately. Instead initially the new cell block shares the squares with old one, but after $k+2$ steps they have moved to the right, displacing their neighbors to the right, and so on. Hence even in this case the time needed is at most proportional to the number of coding automata.

It is a straightforward exercise to check that for each phase the time and the maximum number of finite automata existing simultaneously in a configuration satisfy the bounds given in the lemma. ∎

Now we are ready to give the

**Proof of Theorem 8.1.** Since one can argue similar to the proof of Theorem 7.1 we confine ourselves to the description of a PTM $D$ witnessing the properness of the inclusion.

For an input word $w \in A^+$ $D$ works in 4 phases:

1. First $D$ checks whether $w$ has the form $uv$ where $u$ is the coding of an PTM $P$ with the correct number of input symbols and $v \in \{\texttt{[]}\}^*\{\lambda, \texttt{]}\}$. If this is not the case, $D$ rejects $w$.

2. Using the assumptions about $s$, $t$ and $h$ the values $t(|w|)$ and $h(|w|)$ are computed and stored in chains of finite automata. The ends of a tape segment of length $s(|w|)$ are marked and, starting at the same left end, a tape segment of $h(|w|)$ squares. If it would be necessary for a finite automaton to leave the longer segment during the simulation, $w$ is rejected.

3. Then $D$ tries to generate the shortest coding of the configuration $c_w$ of $P$.

4. Finally $D$ works like the universal simulator described in the proof of Lemma 8.2 above. In addition before phase 5 it is always checked, whether the number of marking automata is at most $3h(|w|)$. If this is not the case, the simulation is stopped and $w$ is rejected. Parallel to the simulation in each step the counter which has been initialized with $t(|w|)$ is decremented by 1.

   $D$ stops the simulation whenever $P$ reaches a final configuration or the counter has been decremented to 0. If $P$ would accept $w$, then $D$ rejects it. If $P$ would reject $w$ or if the simulation was stopped without reaching a final configuration of $P$, then $D$ accepts $w$.

The rest of the proof is analogous to that one in the previous section. ∎

Finally let us have a look at an implication of a collapse of the time hierarchy of CA with space complexity $n$ for PTM. From Theorems 5.1 and 8.1 one can deduce:

**8.3 Corollary.** *If* CA–ST$(n, 2^n)$ = CA–ST$(n, n)$ *then for any* $b \geq 2$ *holds:*

$$\begin{aligned}\text{PTM–STPA}(n+1, \frac{2^{n/\log n}}{\log n}, \frac{n}{\log n}, b) \\ \subsetneq \text{PTM–STPA}(n+1, n2^{n/\log n}, n, b) \\ = \text{PTM–STPA}(n+1, n, n, b)\end{aligned}$$

If the polynomial time hierarchy for $n$-space bounded CA collapses, then there are languages which cannot be recognized by PTM in almost exponential time with $n/\log n$ processors but which can be recognized by PTM with $n$ processors in linear time — *if the tape alphabet is fixed.* And it is because of the last remark, that the statement does not contradict Lemma 4.2. In fact in its proof we *did* increase the tape alphabet. Hence it has *not* been proved that the polynomial time hierarchy for $n$-space bounded CA does not collapse.

# 9 Summary and Outlook

The processor complexity of PTM has been used to measure the amount of parallelism in CA algorithms. In the extreme cases PTM degenerate to sequential Turing machines with one head (no parallelism) or to cellular automata (full parallelism).

It can be proved that an increase of one of time or processor complexity by $n^\varepsilon$ while keeping the other complexity fixed leads to a strictly greater recognition power of PTM. This is so even for a *fixed* space complexity (of $n+1$).

For the first time, a hierarchy of complexity classes could be found within the family of languages that can be recognized by cellular automata in polynomial time. Though it is a hierarchy of complexity classes related to PTM *not* having maximum processor complexity. Hence the problem whether the time complexity hierarchy for cellular automata working in real space collapses or not remains open. However, we take the results obtained as an indication that this is unlikely.

Another open problem is the question, what it really means to fix the size of tape alphabets as it has been done in Section 8. The implications of this measure, e.g. concerning the processor complexity if the algorithm has to be kept within some space bound, are not obvious.

## Acknowledgment

The author gratefully acknowledges interesting and helpful discussions with Heinrich Rust and Roland Vollmar.

## References

[1] R. M. Balzer. An 8-state minimal time solution to the firing squad synchronization problem. *Information and Control*, 10:22–42, 1967.

[2] H.-D. Gerken. *Über Synchronisationsprobleme bei Zellularautomaten.* Diplomarbeit, Technische Universität Braunschweig, 1987.

[3] A. Hemmerling. Concentration of multidimensional tape-bounded systems of Turing automata and cellular spaces. In L. Budach, editor, *International Conference on Fundamentals of Computation Theory (FCT '79)*, pages 167–174, Berlin, 1979. Akademie-Verlag.

[4] A. Hemmerling. Systeme von Turing-Automaten und Zellularräume auf rahmbaren Pseudomustermengen. *Journal of Information Processing and Cybernetics EIK*, 15:47–72, 1979.

[5] A. Hemmerling. On the power of cellular parallelism. In Tamás Legendi, Dennis Parkinson, Roland Vollmar, and Gottfried Wolf, editors, *Parcella '86, Third International Workshop on Parallel Processing by Cellular Automata and Arrays, Berlin, September 9-11*, pages 210–217, Berlin, 1986. North-Holland/Akademie-Verlag.

[6] F. C. Hennie. One-tape, off-line Turing machine computations. *Information and Control*, 8:553–578, 1965.

[7] O. H. Ibarra and T. Jiang. On some open problems concerning the complexity of cellular arrays. In J. Karhumäki, H. Maurer, and G. Rozenberg, editors, *Results and Trends in Theoretical Computer Science*, number 812 in LNCS, pages 183–196. Springer, 1994.

[8] O. H. Ibarra, S. M. Kim, and S. Moran. Sequential machine characterizations of trellis and cellular automata and applications. *SIAM Journal on Computing*, 14:426–447, 1985.

[9] J. Mazoyer. On optimal solutions to the firing squad synchronization problem. *Theoretical Computer Science*, 168(2):367–404, 1996.

[10] W. J. Paul. On time hierarchies. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 218–222, 1977.

[11] S. Ruby and P. Fischer. Translational method and computational complexity. In *Proceedings of the Sixth Annual IEEE Symposium on Swithing Circuit Theory and Logical Design*, pages 173–178, 1965.

[12] H. J. Stoß. $k$-Band-Simulation von $k$-Kopf-Turing-Maschinen. *Computing*, 6:309–317, 1970.

[13] T. Suel. *Zur Zustandsänderungskomplexität von Zellularautomaten.* Diplomarbeit, Technische Universität Braunschweig, 1990.

[14] R. Vollmar. On cellular automata with a finite number of state changes. *Computing*, 3:181–191, 1981.

[15] R. Vollmar and Th. Worsch. *Modelle der Parallelverarbeitung – eine Einführung.* Teubner, Stuttgart, 1995.

[16] J. Wiedermann. Five new simulation results on Turing machines. Technical report 631, Academy of Sciences of the Czech Republic, Institute of Computer Science, 1995.

[17] J. Wiedermann. Parallel Turing machines. Technical Report RUU-CS-84-11, University Utrecht, Utrecht, 1984.

[18] J. Wiedermann. Weak parallel machines: a new class of physically feasible parallel machine models. In I. M. Havel and V. Koubek, editors, *MFCS '92, 17$^{th}$ International Symposium Mathematical Foundations of Computer Science*, volume 629 of *LNCS*, pages 95–111. Springer, 1992.

[19] Th. Worsch. *Komplexitätstheoretische Untersuchungen an myopischen Polyautomaten.* Dissertation, Technische Universität Braunschweig, 1991.

[20] Th. Worsch. On parallel Turing machines with multi-head control units. *Parallel Computing*, to appear, 1997.